ANL-20/76

Evaluation of PETSc on a Heterogeneous Architecture, the OLCF Summit System: Part II – Basic Communication Performance

Mathematics and Computer Science Division

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

ANL-20/76

Evaluation of PETSc on a Heterogeneous Architecture, the OLCF Summit System: Part II – Basic Communication Performance

Prepared by
Junchao Zhang, Richard Tran Mills, and Barry Smith
Mathematics and Computer Science Division, Argonne National Laboratory

October 2020
This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Evaluation of PETSc on a Heterogeneous Architecture, the OLCF Summit System: Part II – Basic Parallel Communication Performance

Junchao Zhang, Richard Tran Mills, Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory

Abstract

Nearest-neighbor communication is at the heart of many high-performance parallel computations. We report on the performance of such communication on the Oak Ridge Leadership Computing Facility system Summit in the context of the PETSc communication module. The analysis in this report includes basic Ping-Pong point-to-point communication and regular and irregular nearest-neighbor communication. We evaluated PETSc communication performance in these patterns. We also discussed various synchronization models when using GPU-aware MPI.

1 Introduction

We report on the performance of the Portable, Extensible Toolkit for Scientific Computation (PETSc) [2, 3] communication infrastructure using basic Ping-Pong point-to-point communication and regular and irregular nearest-neighbor communication on the IBM/NVIDIA Summit computing system [11] at the Oak Ridge Leadership Computing Facility (OLCF). This report is a continuation of Evaluation of PETSc on a Heterogeneous Architecture, the OLCF Summit System: Part I – Vector Node Performance [8] that introduces the Summit architecture and analyzes simple on-node performance characteristics. This report builds on the previous report's analysis and will not repeat the detailed material. Part III will continue the analysis in this report for unstructured mesh communication for partial differential equations.

The planned U.S. Department of Energy exascale computing systems [10] have designs similar to that of Summit. Thus, having a well-developed understanding of Summit is essential in order to prepare for these systems. This document is not intended to provide a strict benchmarking of the Summit system; instead, the intention is to develop an understanding of systems similar to Summit in order to guide PETSc development.

2 The Summit System and Experimental Setup

Figure 1 shows the basic communication pathways of a Summit compute node. Each node has two CPU sockets, each containing one IBM Power9 CPU accompanied by three NVIDIA Volta V100 GPUs. The CPUs and GPUs are connected by NVIDIA's NVLink interconnect, which has a bidirectional bandwidth of 50 GB/s. Communication between the two CPUs is provided by IBM's X-Bus, with a bidirectional bandwidth of 64 GB/s. Each CPU also connects to a single Mellanox InfiniBand ConnectX-5 (EDR IB) network interface card (NIC) through a PCIe Gen4 x8 bus with a bidirectional bandwidth of 16 GB/s. The NIC has an injection bandwidth of 25 GB/s.

PETSc uses MPI for communication between processes. When data is in the GPU memory, PETSc can copy the data to the CPU memory, perform the communication with regular MPI on the CPUs, and then copy the received data to the GPUs. The preferred approach, however, is to use CUDA-aware MPI, with which PETSc can pass GPU address pointers directly to MPI routines. This report focuses on this approach because it provides better performance than does copying the data to the CPU memory. A high-quality CUDA-aware MPI implementation would use NVIDIA's GPUDirect point-to-point (P2P) and remote direct memory access (RDMA) technologies. With GPUDirect P2P, data can be directly copied between the memories of two GPUs within a node. With GPUDirect RDMA, GPUs can communicate directly to the

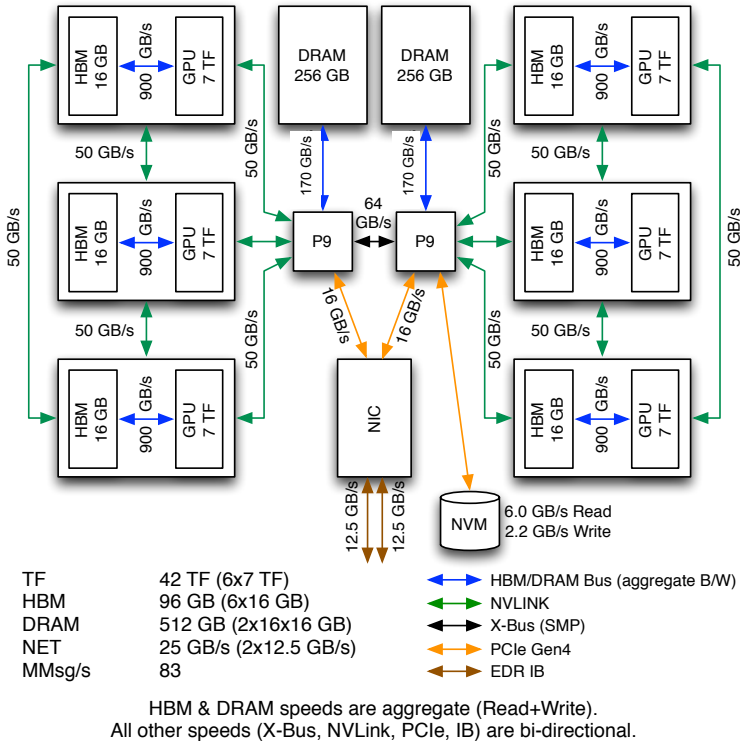

Figure 1: Diagram of a Summit node with communication pathway[6]. There are two IBM Power9 CPUs (P9), with each attached to three NVIDIA V100 GPUs.

NIC and send or receive data without staging in the CPU memory. The former is useful in MPI intranode communication, and the latter is useful in MPI inter-node communication.

The NIC that connects the node to the parallel network is connected to a programmable "local network" that connects it to the CPU memory as well as the GPU memory. This combination of interconnects means that the parallel communication latency and bandwidth (see the first report [8]) are limited by the NIC, the local network, the NVLinks from the CPU to the local network, and the GPU memory but not the CPU memory. However, CUDA-aware MPI calls (send, receive, and waits) must be called by code running on the CPU cores. Ongoing research focuses on triggering the MPI communication from within CUDA kernels to avoid the extra CPU-to-GPU operations, but this capability is not currently available. The total communication time is a combination of the physical/software latencies and bandwidths of the various hardware components plus the latencies and bandwidths induced by the software stack.

3 MPI Point-to-Point Latency on Summit

In [7], the authors evaluated MPI point-to-point latency and bandwidth on a GPU-enabled OpenPower system similar to Summit, using three MPI implementations: MVAPICH2-GDR, OpenMPI, and IBM Spectrum MPI. In this section, we repeat their latency experiments on Summit. We use only Spectrum MPI [1] since it is the only supported MPI on the machine; the others are difficult to install and use. Measuring MPI performance on Summit is not the purpose of this report. What we want to know is what communication performance PETSc can provide, since PETSc users, and PETSc code itself, usually do not directly call MPI; instead, they access it through PETSc application programming interfaces.

We used osu_latency from the OSU Microbenchmarks 5.6.2 [9], which can measure latency with CPU or GPU buffers. We mainly focus on the GPU case in this report. This test is also known as the MPI Ping-Pong

[1]We used modules spectrum-mpi/10.3.1.2-20200121, cuda/10.1.243 and gcc/6.4.0. There was an environment variable `PAMI_CUDA_AWARE_THRESH` with default 320000. Spectrum MPI used a so-called *staging* approach for messages larger than it, and used GPUDirect otherwise. We found staging generally gave worse results in MPI latency tests, especially in inter-node tests with large messages. So for tests in this report, we unset this environment variable, in other words, we always used GPUDirect.

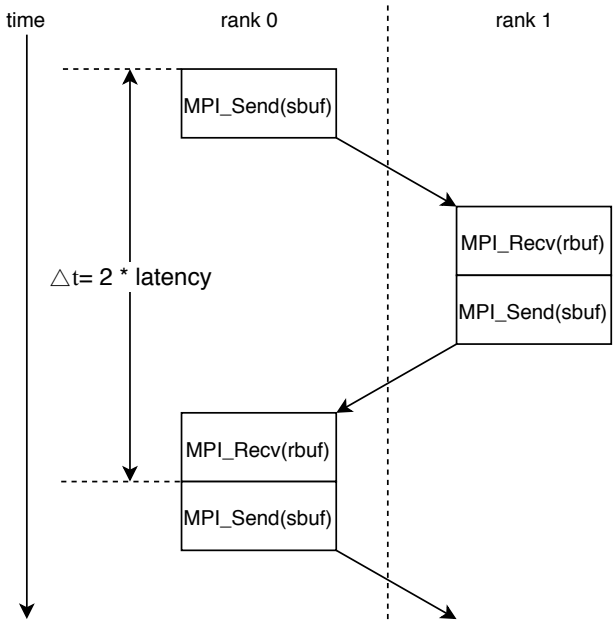

Figure 2: OSU Microbenchmarks latency test [9]

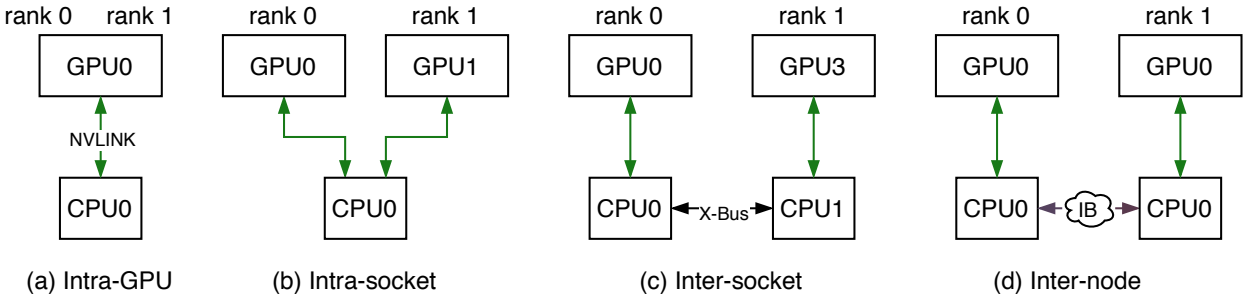

Figure 3: MPI rank placements in the OSU latency test. Unused CPUs/GPUs in each diagram are omitted.

test. Shown in Figure 2, it uses two MPI ranks and allocates a send buffer (`sbuf`) and a receive buffer (`rbuf`) on each rank. Rank 0 `MPI_Send()`s a message of a certain size from its `sbuf` to rank 1's `rbuf`. Once rank 1 `MPI_Recv()`s the message, rank 1 replies with a message of the same size from its `sbuf` to rank 0's `rbuf`. When rank 0 gets the reply, this finishes a round trip from rank 0 to rank 1. The round trip is repeated many times (10,000 times for messages $\leq$ 8 kilobytes and 1,000 times otherwise). The latency is calculated as the average time of a one-way trip. The send and receive buffers are distinct in order to minimize the cache effect, although that is not very important on GPUs, as we will see later. The microbenchmark uses `MPI_Wtime()` for timing and assumes that the send buffers are ready for MPI, so no CUDA synchronizations are involved.

We placed the two MPI ranks on the same GPU, on two GPUs attached to the same CPU, on two GPUs attached to different CPUs within a node, and on two GPUs across nodes. These rank placements are referred to as intra-GPU, intra-socket, inter-socket, and inter-node, respectively, in this report, as shown in Figure 3. First we enabled Spectrum MPI's CUDA support with `jsrun --smpiargs ''-gpu''` and measured device to device Ping-Pong latency with "`osu_latency D D`". Then we modified osu_latency.c and allocated two pinned (page-locked) buffers on host with `cudaMallocHost()` as intermediate send/receive buffers. To send data, we first copied data from GPU to the send buffer on CPU with `cudaMemcpy()`; after receiving the data in the receive buffer on CPU, we copied it to GPU with `cudaMemcpy()` again. This mimics a code without CUDA-aware MPI support. We measured the modified osu_latency again with the same `jsrun` command line. The results are shown in Table 1. Although the microbenchmark can test message sizes starting from 0,

for brevity we omitted results for messages smaller than 8 bytes. The intra-GPU results with CUDA-aware MPI are better than those reported in Figure 6 of [7]. The remaining results largely match with those in Figures 4, 10, and 12 of [7]. We can regard these performance numbers as an upper bound that a similar PETSc benchmark could achieve. We can see the one with CUDA-aware MPI was always better than the one without.

Table 1: MPI Ping-Pong latency[2] measured by osu_latency from the OSU Microbenchmarks [9].

Message size (bytes)	Latency (µs) with CUDA-aware MPI				Latency (µs) without CUDA-aware MPI			
	Intra-GPU	Intra-socket	Inter-socket	Inter-node	Intra-GPU	Intra-socket	Inter-socket	Inter-node
8	20.1	17.8	19.3	6.0	28.73	27.68	28.78	28.76
16	20.1	17.8	19.4	6.0	28.71	27.71	28.75	28.75
32	20.1	17.8	19.4	6.8	28.75	27.74	28.76	29.42
64	20.1	17.8	19.5	6.0	28.79	27.79	28.87	28.82
128	20.1	17.8	19.5	6.1	28.79	27.81	28.88	29.60
256	20.1	17.8	19.4	6.2	28.93	27.96	29.18	29.54
512	20.1	17.8	19.5	6.2	29.05	28.13	29.39	30.20
1K	20.1	17.8	19.4	6.3	29.26	28.40	29.63	30.18
2K	20.0	17.8	19.4	6.8	29.61	28.89	30.03	31.45
4K	20.1	17.8	19.4	7.2	30.65	29.63	31.51	32.37
8K	20.1	17.8	19.5	8.2	31.31	30.51	32.41	35.24
16K	20.1	17.8	19.5	9.3	34.54	33.84	35.67	38.28
32K	20.0	17.8	19.4	11.4	33.17	31.51	34.65	37.94
64K	20.1	18.5	20.1	14.1	35.72	34.37	38.99	43.81
128K	20.1	20.0	21.6	19.9	41.81	40.03	47.24	50.62
256K	20.1	22.6	24.6	30.5	53.32	50.95	63.07	67.03
512K	20.4	28.2	30.9	51.8	74.10	72.28	94.70	100.05
1M	20.7	39.4	43.2	98.2	115.54	114.08	156.93	169.05
2M	25.6	61.7	68.2	191.2	199.58	196.53	280.78	305.03
4M	31.6	106.6	140.9	436.7	389.26	370.69	532.59	583.37

For a message of s bytes, its MPI Ping-Pong latency l can be modeled as

$$l = \alpha + \beta s, \tag{1}$$

where α is the start-up time and β is reciprocal of the `MPI_Send/Recv()` bandwidth. The term *latency* has two usages in communication; it is used as the total time of communication, but in some literature it also is used as the start-up time α. We follow the former usage and hope it will not be confusing in the document. We fit the data in Table 1 with the least squares algorithm and obtained the start-up time and bandwidth for various rank placements, shown in Figure 4.

From the computed results, the intra-GPU bandwidth reaches 83.4% of half of the GPU peak memory bandwidth of 900 GB/s (note that we both read and write the same GPU memory in this case). The intra-socket and inter-socket bandwidths reach 94.6% and 71.8% of the NVLink bandwidth at 50 GB/s, respectively, while the inter-node one reaches only 40.4% of the EDR IB bandwidth at 25 GB/s. Since GPU virtualization on Summit comes with some cost, up to 20%, we highly recommend using one MPI rank per physical GPU. In the subsequent studies in this report, we follow this approach.

4 The Communication Module in PETSc

4.1 Introduction

`PetscSF` is PETSc's communication module. It is heavily used by other PETSc modules internally. Applications can also call it directly. `VecScatter`, a public interface for communication on PETSc vectors, is implemented by utilizing `PetscSF`. `PetscSF` abstracts communications into a star-forest graph. A star-forest

[2]We note that we observed considerable variations across runs (jobs) in the inter-node big messages tests (e.g., 4 MB), which could be 20% higher than what is reported here. We think this is due to the location of the two nodes in the communication network allocated by the job submission system. Two-node results in this report were got from nodes with names like `d36n09` and `d36n10`. Telling from names, we believe they were physically very close.

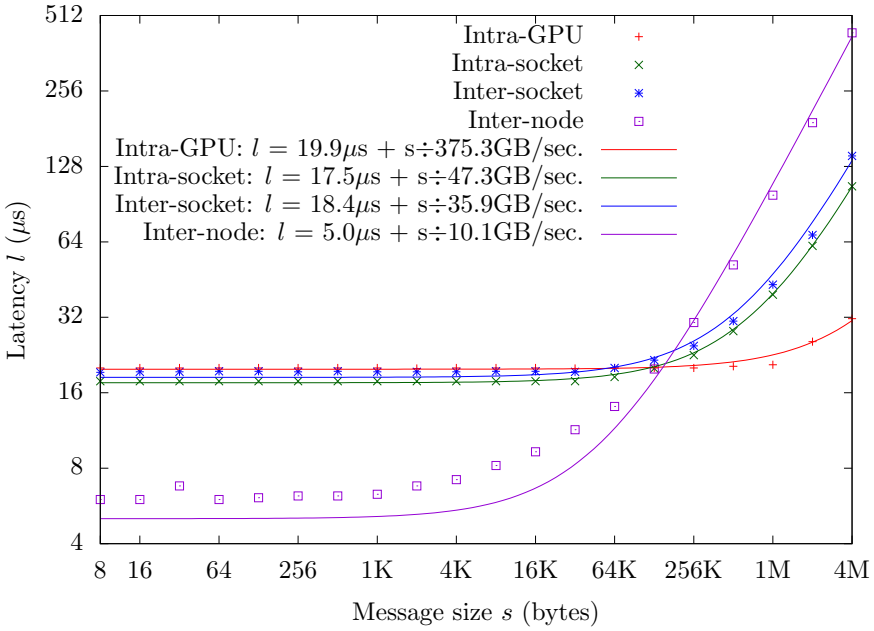

Figure 4: MPI Ping-Pong latency modeled by $l = \alpha + \beta s$ for various rank placements using data in Table 1. The curves are fit by the linear least squares method.

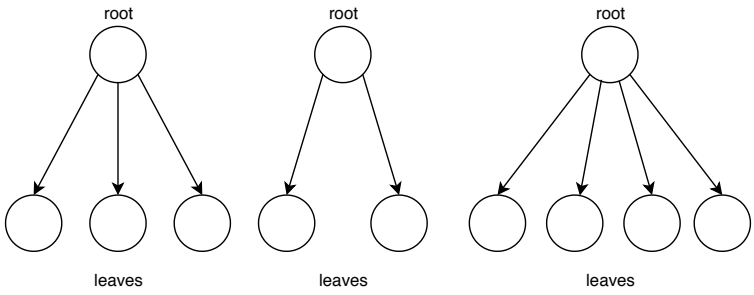

Figure 5: Star-forest example

is a forest containing multiple star-shaped trees, where each tree has a height of one, with one root and multiple leaves. See Figure 5 for example.

To build a `PetscSF`, users need to provide on each MPI process two integer-indexed spaces: the leaf space and the root space. Leaves in the leaf space can be dense (i.e., contiguous) or sparse and must be local to the process such that an integer can identify a leaf. Roots must be dense. Roots might be remote; in that case one uses `rank`, `index`) pairs to specify roots that the local leaves connect to, where rank is the MPI `rank` a root resides in and `index` is the *local* index of the root on that MPI rank.

`PetscSF` provides split-phase routines to communicate between roots and leaves of a star-forest. For example, `PetscSFReduceBegin/End()` reduces leaves to their connected roots with a given MPI reduction operation. `PetscSFBcastBegin/End()` broadcasts roots to their connected leaves. Users are able to put computation between `PetscSFXxxBegin/End()` so that communication and computation can be overlapped. In addition, one can interleave communications on the same `PetscSF` with different leaf data or root data.

4.2 PetscSF Implementation

A naive implementation of a star-forest would post sends and receives for every edge in the star-forest. This is inefficient, so `PetscSF` has a setup phase, which performs an index analysis to agglomerate messages and also to provide hints for other optimizations. The index analysis cost is low and can be amortized by repeated calls to the `PetscSF` communication routines.

On each MPI process, PetscSF internally computes the process's neighbors (a list of destination ranks and source ranks) with which the process will communicate, in other words, send data to or receive data from. For each destination, the process computes the indices of the local data (leaves or roots depending on the context) it needs to send. For each source, it computes the indices that indicate the locations where it should deposit the received data. When a neighbor is the process itself, we call the communication *local* communication; otherwise we call it *remote* communication. We separate local and remote communications since for the local communication we can bypass MPI and enjoy unique optimization opportunities.

For remote communication, PetscSF allocates on each MPI process a send buffer and a receive buffer. Consider the example `PetscSFReduceBegin(sf,unit,leafdata,rootdata,op)`. A process packs selected entries of leafdata into the send buffer and then sends them. After a process receives data in the receive buffer it unpacks entries from the buffer and deposits them to rootdata. Each remote neighbor has its own region in the send and receive buffers. PetscSF's pack/unpack routines are overloaded according to the location of the root/leafdata. When data is in the CPU memory, the routines are CPU functions; when data is in GPU memory, the routines are CUDA kernels, where each CUDA thread works on a leaf/root. PetscSF uses atomic instructions in the `unpack` CUDA kernels when there are chances of data race conditions.

PetscSF may use index analysis results to exploit optimizations in order to decrease the packing cost. For instance, in `PetscSFReduce()`, when the leaf indices used in packing are contiguous, PetscSF aliases leafdata as the send buffer and completely avoids the packing. An obvious question is whether in `PetscSFReduce()`, when the root indices for unpacking are contiguous, it can alias rootdata as the receive buffer and avoid the unpacking. The answer depends on the reduction operation argument `op`. If op is `MPI_REPLACE`, it can; otherwise, it cannot and has to allocate a receive buffer and launch an `unpack` kernel to perform the reduction. Even in this case, it has an optimization when the root indices are contiguous. It can skip copying the unneeded indices to the GPU and use a simpler expression in the `unpack` kernel. PetscSF employs persistent `MPI_Isend/Irecv()` routines for communication. With this and buffer aliasing, this means that in an SF's lifetime it may encounter different send/receive buffers. PetscSF handles this complexity with MPI persistent requests. PetscSF does buffer allocation and MPI persistent request initialization on demand and thus uses the resources only when needed.

PetscSF treats local communication as a scatter operation: `x[idx[i]]` $\rightarrow$ `y[idy[i]]`, for `i` $\in$ [0, n). The scatter is a GPU kernel when the data is on the GPU. It uses simpler expressions such as `x[startx+i]` $\rightarrow$ `y[idy[i]]` when it knows the indices. Other variants exist, such as when the scatter is a memory copy, or even a no-op when it determines it is a memory copy with the same destination as the source. PetscSF exploits these opportunities to optimize local communication.

In `PetscSFXxxBegin()`, it first checks memory types of the input rootdata and leafdata, to determine whether they point to CPU or GPU memory. It needs this information to set up the needed data structures and select the appropriate pack routines. Then it posts `MPI_Irecv()` requests through `MPI_Startall()`, calls a pack routine to pack source data into the send buffer, and posts `MPI_Isend` requests. After that, it calls a scatter routine to do local communication. In `PetscSFXxxEnd()`, it waits for the requests it has posted with `MPI_Waitall()`. It then calls an unpack routine to unpack the data from the receive buffer. The pack/unpack is skipped sometimes, as discussed above.

Although PetscSF has a "copy to CPU memory" fallback mode for systems without GPU-aware MPI, we focus exclusively on the code path using GPU-aware MPI since it avoids the back-and-forth buffer copying between CPUs and GPUs and has superior performance.

CUDA kernels are executed asynchronously with respect to the CPUs. CUDA provides functions such as `cudaDeviceSynchronize()` and `cudaStreamSynchronize()` for users to synchronize the whole device or just a CUDA stream. They are blocking calls on CPU threads. The CUDA driver reserves a region of pinned (i.e., pages-locked) host memory as a shared sync location between the CPU and the GPU, where it can store GPU progress values. Every major GPU operation is followed by a command to write the new progress value to the shared sync location through direct memory access. By checking the progress value, these functions can know whether the previously issued GPU operations are completed [12].

When a PetscSF routine is called, the leaf/root data might still be in the process of being computed by a CUDA kernel on a CUDA stream that is not known to the PetscSF. Therefore, for correctness, PetscSF must call `cudaDeviceSynchronize()` to wait for the data to be ready. PetscSF could launch `pack/unpack` kernels on its own stream. On the sender side, PetscSF calls `cudaStreamSynchronize()` on the stream before `MPI_Isend()`. On the receiver side, after `MPI_Waitall()`, PetscSF is assured that the data has been received so it launches the `unpack` kernel immediately and then calls `cudaStreamSynchronize()` to make

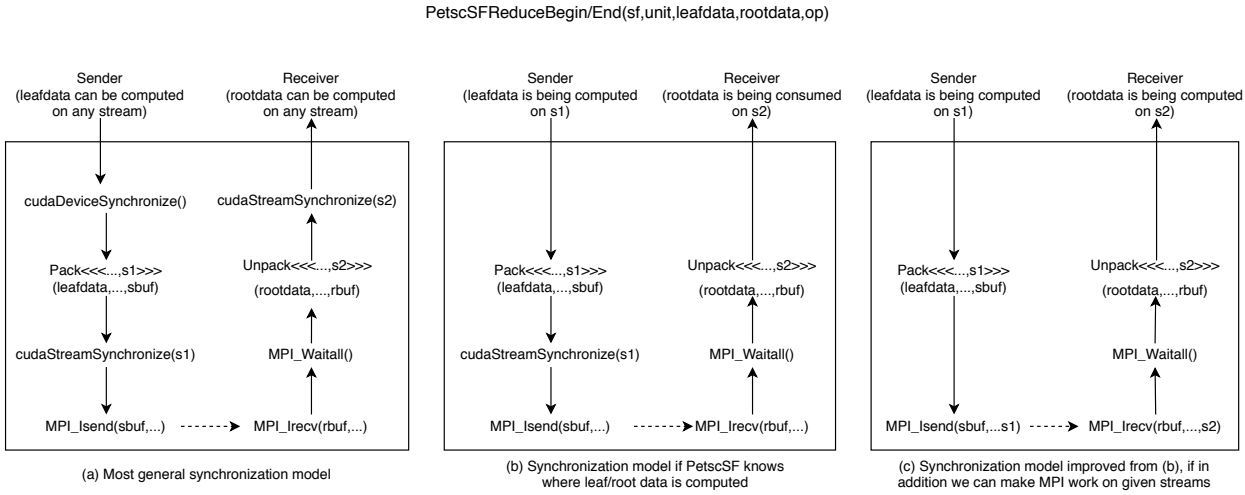

Figure 6: Different synchronization models in `PetscSF`

the data ready for the `PetscSF` clients (either applications or other modules of PETSc). This procedure is demonstrated in Figure 6(a) using `PetscSFReduce()` as an example. Many synchronizations are involved. If `PetscSF` knew the streams where the leaf/rootdata was produced or is to be consumed, it could eliminate the synchronizations before `pack` and after `unpack`, as shown in Figure 6(b). Furthermore, if the MPI routines were CUDA-stream aware, for example, by taking a stream argument or other means, and worked like a kernel launch, we then could remove the synchronization before `MPI_Isend()`, as shown in Figure 6(c). Doing so, however, requires support from MPI that is currently not available; see the MPI and CUDA semantic mismatch discussion in [5].

Model (a) is the most general model. Since PETSc currently uses only the CUDA default stream, we provide an option `-sf_use_default_stream` to allow `PetscSF` to skip the `cudaDeviceSynchronize()` call before the `pack` and the `cudaStreamSynchronize()` call after the `unpack`. This option turns Model (a) into Model (b) in Figure 6 (with `s1 = s2 = NULL`). For the performance experiments, we also provide an option `-sf_use_stream_aware_mpi` that pretends that the underlying MPI knows the streams where the send/receive data is being produced/consumed and eliminates the `cudaStreamSynchronize()` after the `pack` and turns Model (b) into Model (c).

5 Experimental Results

5.1 PetscSF without pack/unpack

We begin with a Ping-Pong test that uses `PetscSF` but otherwise has the same parameters as those in the OSU microbenchmark Ping-Pong test used in Section 3. Suppose we want to measure latency for a message of $8n$ bytes. We build a `PetscSF` in which rank 0 has n roots and zero leaves, while rank 1 has 0 roots and n leaves, as shown in Figure 7. Rank 1's leaves are one-on-one sequentially connected to rank 0's roots. With this `PetscSF`, `PetscSFBcast()` will send from rank 0 to rank 1, while `PetscSFReduce()` will send from rank 1 to rank 0. We used double precision and `MPI_DOUBLE` as the MPI datatype for roots and leaves. For the study, we built different `PetscSF`s for the different message sizes. The following loop shows a Ping-Pong test for a given message size. Note that `sbuf` and `rbuf` in the code work as a pair of rootdata on rank 0, and as a pair of leafdata on rank 1, which is intended to mimic the behavior in the OSU microbenchmark.

Since in this test the root/leaf indices are contiguous and we do not perform reductions on the roots, `PetscSF` has optimizations that directly use `sbuf` or `rbuf` as MPI's send/receive buffers and avoid the packing/unpacking kernels. In other words, we get the simplified code path depicted in Figure 8(a). To remove the `cudaDeviceSynchronize()` before the `MPI_Isend`, we use option `-sf_use_default_stream` to indicate that the root/leaf data is available on the default stream, and we obtain the code path shown in Figure 8(b). The `cudaStreamSynchronize(NULL)` is there because the condition that leaf data is on

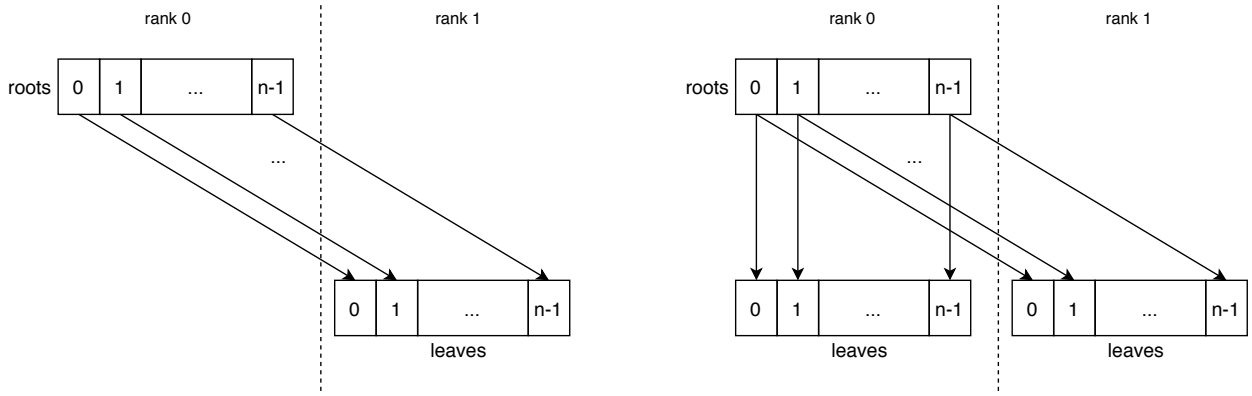

Figure 7: Star-forests in the `PetscSF` Ping-Pong and `unpack` tests (left) and in the `PetscSF` scatter test (right)

```
for (i=0; i<niter; i++) {
  PetscSFBcastBegin(sf,MPI_DOUBLE,sbuf,rbuf);
  PetscSFBcastEnd(sf,MPI_DOUBLE,sbuf,rbuf);
  PetscSFReduceBegin(sf,MPI_DOUBLE,sbuf,rbuf,MPI_REPLACE);
  PetscSFReduceEnd(sf,MPI_DOUBLE,sbuf,rbuf,MPI_REPLACE);
}
```

Listing 1: sf_pingpong benchmark loop

the default stream does not necessarily mean it is ready for MPI to send. To remove it, we use option `-sf_use_stream_aware_mpi` to indicate that MPI knows which streams to use for the input or output data. Although the IBM Spectrum MPI does not support this feature, it does not matter in this simple test since the input data is always ready and the test does not use the output data. This produces the code path in Figure 8(c).

We measured the intra-socket GPU to GPU latency for the three variants. The results are shown in columns Opt-A/B/C, respectively. Comparing intra-socket columns Opt-A and Opt-B, we can see that `cudaDeviceSynchronize()` has a slightly higher cost (about 1.5µs) than does `cudaStreamSynchronize()`. Comparing intra-socket columns Opt-B and Opt-C, we know the time of a `cudaStreamSynchronize()` call is about 4µs, since Opt-C does not have any synchronization. We profiled the code with Opt-C and found the time-consuming CUDA driver routine `cuPointerGetAttribute()`, which was called twice in `PetscSFXxxBegin()` to test the pointer attributes (if they point to CPU or GPU memory) of the argument's rootdata and leafdata. Since we knew in this test they were GPU pointers, we manually modified the `PetscSF` code and bypassed the CUDA driver call. The results are in column Opt-D. Comparing it with the intra-socket column in Table 1, we can see that the minimal overhead of PetcSF is around 1µs over pure MPI, which is satisfying. Overall the `PetscSF` Ping-Pong latency is about 6µs longer than that of pure MPI. For completeness, Table 2 also shows inter-socket and inter-node latency with Opt-B, which is PETSc's default model; we will use it for the remaining tests in this report. We also modeled `PetscSF` Ping-Pong latency with Opt-B using the linear model Eq. 1 in Section 3, shown in Figure 9. Compared with data in Figure 4, PetcSF Ping-Pong had a longer start-up time but the same bandwidth. Comparing the most general synchronization model in Figure 6(a) with PETSc's default model in Figure 6(b), we see that the former has one `cudaDeviceSynchronize()` and one `cudaStreamSynchronize()`, whose cost is about 9µs in total, based on the above analysis.

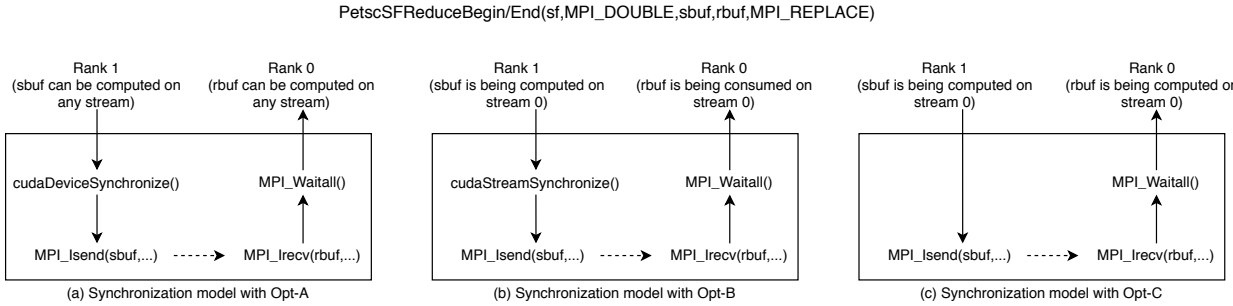

Figure 8: Code paths in the sf_pingpong test with different synchronization models

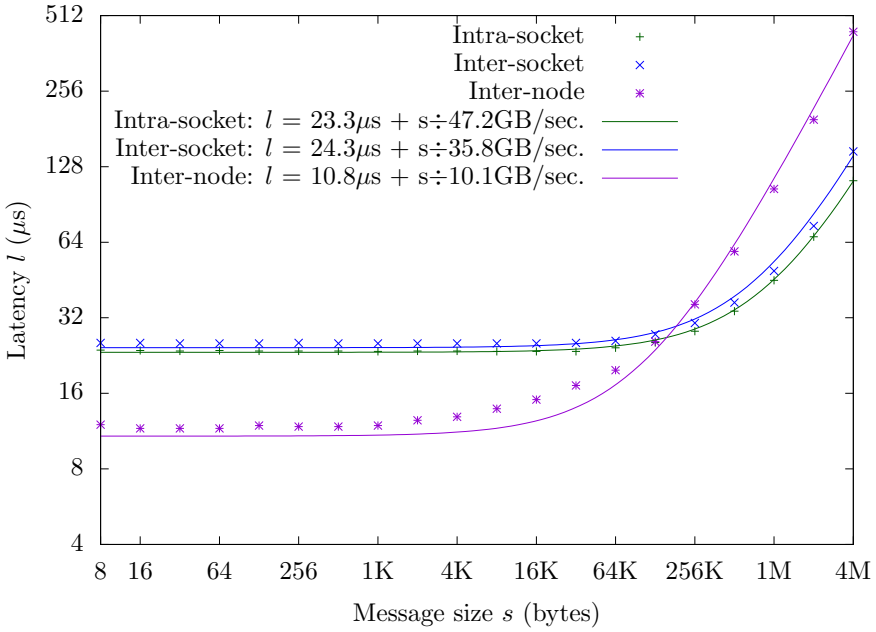

Figure 9: PetscSF Ping-Pong latency modeled by $l = \alpha + \beta s$ for various rank placements using data in Opt-B columns of Table 2. The curves are fit by the linear least squares method.

Table 2: sf_pingpong latency. Options used: Opt-A = `-use_gpu_aware_mpi`; Opt-B = Opt-A + `-sf_use_default_stream`; Opt-C = Opt-B + `-sf_use_stream_aware_mpi`; Opt-D = Opt-C + manually set types of root/leafdata as GPU memory pointers. PETSc's default is Opt-B.

Message	Intra-socket latency (µs)				Latency (µs) with Opt-B	
size (bytes)	Opt-A	Opt-B	Opt-C	Opt-D	Inter-socket	Inter-node
8	25.3	23.8	19.9	19.0	25.4	12.0
16	25.2	23.7	19.7	19.0	25.4	11.6
32	25.2	23.6	19.7	18.9	25.3	11.6
64	25.2	23.7	19.7	19.0	25.3	11.6
128	25.2	23.6	19.8	19.0	25.3	11.9
256	25.2	23.6	19.8	19.0	25.4	11.8
512	25.2	23.6	19.8	19.0	25.3	11.8
1K	25.2	23.5	19.8	19.0	25.3	11.9
2K	25.2	23.6	19.8	19.0	25.3	12.5
4K	25.1	23.6	19.8	19.0	25.3	12.9
8K	25.0	23.5	19.6	18.9	25.3	13.9
16K	25.3	23.5	19.8	18.9	25.3	15.1
32K	25.3	23.5	19.8	19.0	25.4	17.2
64K	25.7	24.3	20.5	19.7	25.9	19.8
128K	27.3	25.5	21.7	20.9	27.5	25.7
256K	30.0	28.3	24.5	23.6	30.5	36.2
512K	35.5	34.0	30.1	29.3	36.8	58.8
1M	46.8	45.1	41.3	40.5	49.2	104.3
2M	68.9	67.3	63.6	62.8	74.3	197.0
4M	113.9	112.5	108.6	107.9	147.2	441.2

```
for (i=0; i<niter; i++) {
  PetscSFBcastAndOpBegin(sf,MPI_DOUBLE,rootdata,leafdata,MPI_SUM);
  PetscSFBcastAndOpEnd(sf,MPI_DOUBLE,rootdata,leafdata,MPI_SUM);
  PetscSFReduceBegin(sf,MPI_DOUBLE,leafdata,rootdata,MPI_SUM);
  PetscSFReduceEnd(sf,MPI_DOUBLE,leafdata,rootdata,MPI_SUM);
}
```

Listing 2: sf_unpack benchmark loop

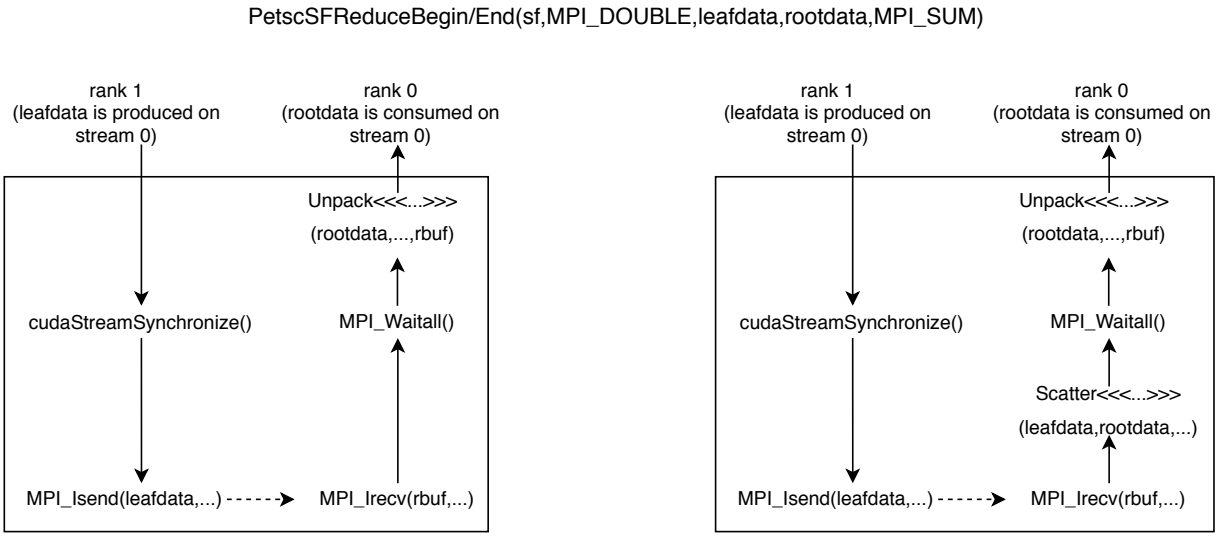

Figure 10: Code paths of `PetscSFReduce()` in tests sf_unpack (left) and sf_scatter (right)

5.2 PetscSF with unpack and local communication

We now turn to the `unpack` kernels and local communications. We slightly modified the sf_pingpong test and created a new test called *sf_unpack*. In sf_unpack we used only one set of root data on rank 0 and one set of leaf data on rank 1. We added roots to leaves with `PetscSFBcastAndOp()` and leaves to roots with `PetscSFReduce()` using the code in Listing 2. Because of the use of `MPI_SUM`, we need a receive buffer at the destination and an `unpack` kernel to perform the addition. With PETSc's default option, we have the code path shown in Figure 10. Comparing it with Figure 8(b), we see that we pay an extra cost for calling `unpack`, which includes the kernel launch time and kernel execution time.

To see the local communication behavior, we created another test called *sf_scatter* by merely changing the `PetscSF`s used in the sf_unpack test. We added leaves on rank 0 and made them connected to its roots one-on-one. An example `PetscSF` is shown on the left of Figure 10. With the new `PetscSF`s and the same code in Listing 2, `PetscSFBcastAndOp()` will add roots on rank 0 to both local and remote leaves; and `PetscSFReduce()` will add both the local and remote leaves to the roots. The code path for `PetscSFReduce()` is shown in the right of Figure 10. On rank 0, the local communication is done through the `scatter` kernel, which directly works on rootdata and leafdata. The remote communication is done through the `unpack` kernel, which works on rootdata and the receive buffer `rbuf`. The two kernels are executed in the default stream one after another, so we are not concerned with the data-race condition in the reduction. Also note that `scatter` is called between `MPI_Irecv()` and `MPI_Waitall()`, so that local communication is overlapped with remote communication.

For a fair comparison, we modified sf_pingpong to let it use one set of root/leaf data (the code is equal to replacing `MPI_SUM` in Listing 2 with `MPI_REPLACE`) and called it *sf_newpingpong*. We tested sf_newpingpong, sf_unpack, and sf_scatter, and we present their latency $l_{pingpong}$, l_{unpack}, and $l_{scatter}$ in columns of Table 3. Let us denote a kernel K's launch and execution time as $T_l(K)$ and $T_e(K)$, respectively. We make th4e following observations.

1. The results of sf_pingpoing in Table 2 (columns labeled with Opt-B) and the results of sf_newpingpong

in Table 3 are close except for the inter-socket and inter-node tests with large messages. For example, in the inter-node 4 MB message size tests, sf_newpingpong is about 13% faster than is sf_pingpoing. This implies that caching does play a role in these cases. Further investigation is out of the scope of this report.

2. In these tests the roots and leaves are dense so that the `unpack` and `scatter` kernels are a vector addition. Using the GPU memory bandwidth 900 GB/s given in Figure 1, we can get a rough estimation of the kernel's `unpack` and `scatter`'s maximal execution time at 4 MB messages size, $T_e(unpack) = T_e(scatter) = 4$ MB*2/900 GB/s = 9.3µs, including both the read and write times.

3. According to the code path of sf_unpack in the left of Figure 10, its latency can be expressed as

$$l_{unpack} = l_{pingpong} + T_l(unpack) + T_e(unpack). \tag{2}$$

Looking at the first row of Table 3 at the message size of 8 bytes (i.e., one double), if we deem $T_e(unpack) = 0$, then we can easily get kernel launch time $T_l(unpack) = l_{unpack} - l_{pingpong} = 12\mu s$, which generally can also be used as the launch time of other kernels.

4. According to the code path of sf_scatter in the right of Figure 10, its latency can be expressed as

$$l_{scatter} = max\big(l_{pingpong}, T_l(Scatter) + T_e(Scatter)\big) \oplus T_l(unpack) + T_e(unpack). \tag{3}$$

The $\oplus$ indicates that if MPI finishes earlier than the `scatter` kernel, launch of `unpack` can overlap with execution of `scatter`. In our tests, however, one can check that $l_{pingpong} \geq T_l(scatter)+T_e(scatter)$ for almost all cases, such that sf_scatter's latency $l_{scatter} = l_{pingpong} + T_l(unpack) + T_e(unpack) = l_{unpack}$. We can observe that it holds for messages from 8 B to 2 MB. Data for the message size 4 MB is an outlier. We guess that the reason is that the local communication (i.e., the `scatter` kernel) and the remote communication interfere on the memory system, which makes $l_{scatter}$ longer than l_{unpack}. Figure 11 shows the timeline of sf_scatter on rank 0 with message size 4 MB using the NVIDIA profiling tool nvprof. We can clearly see that the execution of the `scatter` kernel is overlapped with MPI communication.

Table 3: One-way latency for the three tests: sf_newpingpong, sf_unpack and sf_scatter

Message	Intra-socket(µs)			Inter-socket(µs)			Inter-node(µs)		
size (bytes)	new Ping-Pong	unpack	scatter	new Ping-Pong	unpack	scatter	new Ping-Pong	unpack	scatter
8	24.3	35.9	35.8	25.4	37.6	37.8	12.2	22.9	23.0
16	24.2	35.7	35.6	25.5	37.5	37.6	11.5	22.6	22.6
32	24.1	35.8	35.8	25.4	37.5	37.8	11.6	22.6	22.8
64	24.2	35.8	35.8	25.4	37.6	37.8	11.6	22.6	22.6
128	24.1	35.7	35.6	25.4	37.5	37.6	11.7	22.8	22.6
256	24.2	35.8	35.8	25.5	37.6	37.8	11.7	22.7	22.7
512	24.2	35.7	35.8	25.4	37.6	37.9	11.8	22.8	23.2
1K	24.2	35.7	35.6	25.4	37.6	37.7	11.9	23.0	22.9
2K	24.2	35.6	35.8	25.4	37.6	37.8	12.5	23.3	23.5
4K	24.1	35.7	35.8	25.4	37.6	37.7	12.9	24.0	23.9
8K	24.0	35.7	35.6	25.6	37.6	37.6	13.8	24.7	25.0
16K	24.0	35.7	35.8	25.6	37.6	37.8	15.0	25.9	25.9
32K	24.1	35.7	35.7	25.7	37.6	37.5	17.2	28.1	28.1
64K	24.7	36.3	36.2	26.3	37.9	38.1	19.8	31.1	31.1
128K	25.9	37.4	37.4	27.7	39.5	39.7	25.5	36.8	36.9
256K	28.5	40.3	40.4	30.7	42.7	42.9	36.2	47.5	47.5
512K	34.2	46.7	46.7	36.9	49.8	49.7	57.5	69.6	69.3
1M	45.3	58.0	58.1	49.3	62.4	62.5	106.5	115.9	115.9
2M	67.6	81.2	81.2	74.0	88.0	88.0	197.5	210.7	210.9
4M	112.2	138.8	140.5	123.5	153.4	160.8	382.7	415.7	427.1

[4]The actual kernel names are `d_ScatterAndXxx` and `d_UnpackAndXxx`, as shown by nvprof. For brevity, we just call them `scatter` or `unpack` in this report.

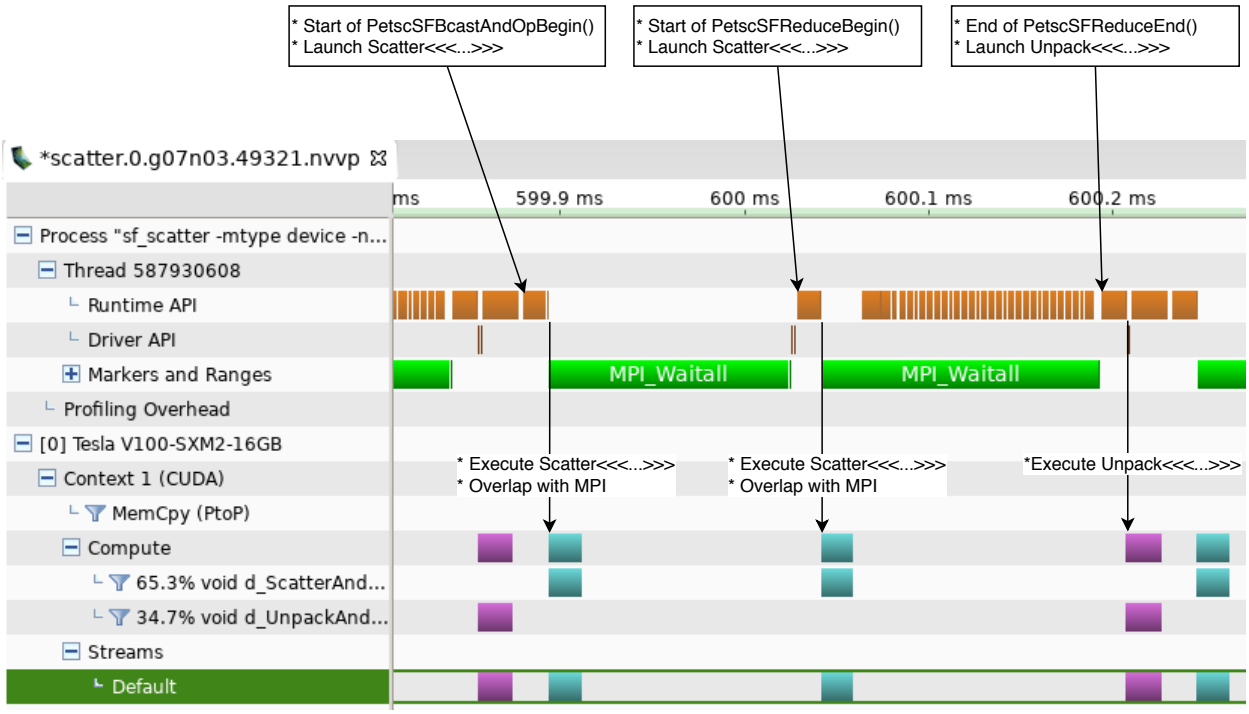

Figure 11: Timeline of one iteration of sf_scatter on rank 0 with 4 MB messages. Local communication (i.e., the `scatter` kernel[4]) is fully hidden by remote communication (i.e., `MPI_Waitall()`).

5.3 PetscSF in regular neighborhood communication

In this section we evaluate `PetscSF` with a five-point stencil code featuring regular neighborhood communication. We construct a two-dimensional grid with `DMDACreate2d()` and then do communication between global vectors and local vectors created with this `DM`. The code creating the `DM` and the vectors is

```
DMDACreate2d(comm,DM_BOUNDARY_PERIODIC,DM_BOUNDARY_PERIODIC,DMDA_STENCIL_STAR,3*n,3*n
    ,3,3,1,1,0,0,&da);
DMCreateGlobalVector(da,&g);
DMCreateLocalVector(da,&l).
```

Here, we create a 3×3 processor grid; we set the stencil type to `DMDA_STENCIL_STAR`, the stencil width to 1, and the boundary type to `DM_BOUNDARY_PERIODIC`; and we let every process have a square subgrid of size $n \times n$. The `DM` is shown on the left of Figure 12. With this setup, each MPI rank will have four neighbors; each will communicate the same amount of data.

In PETSc, global vectors on this grid have a local size of n^2, and elements of the vectors are consecutively stored on each process. The local vectors have a size of $(n+2)^2$, including a ghost (halo) region. `DMGlobalToLocal()`, which is implemented by using `PetscSFBcast()`, copies the local part of a global vector to the interior part of a local vector on each rank and also copies the ghost point values received from neighbors to the halo region of the local vector, shown in the right of Figure 12. Copying the interior region is the local communication, and send/receiving the ghost point values is the remote communication. Each process has to pack four faces of its subgrid into a send buffer and then to its four neighbors and then unpack the ghost point values from its receive buffer. To copy local vectors to global vectors, one uses `DMLocalToGlobal()`, which simply reverses the process above and is implemented by `PetscSFReduce()`.

We can easily see that the local indices of global vectors are contiguously running from 0 to $n^2 - 1$. However, the indices of the ghost points as a whole, or the indices of points in the interior region of a local vector, are not contiguous. Since no hints are given to `PetscSF` that these indices are incidental to a regular 2D grid, a naive implementation would copy the indices to the GPU and resort to in-directions such as `buf[i] = x[idx[i]]` to do the copying. Instead, our optimized `PetscSF` uses index analysis to determine whether the indices associated with a destination rank can be arranged in a 3D subgrid. Suppose we have a

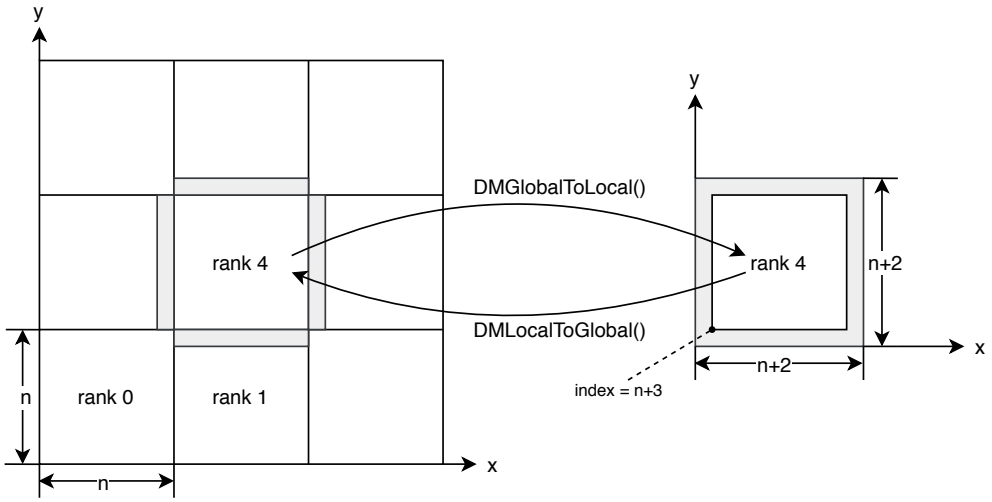

Figure 12: DM created by DMDACreate2d() on nine processors (left) and a local vector on rank 4 (right). Grid points on each processor are numbered in x, y order. Shadowed areas are ghost points.

```
for (i=0; i<niter; i++) {
  DMGlobalToLocalBegin(da,g,INSERT_VALUES,l);
  DMGlobalToLocalEnd(da,g,INSERT_VALUES,l);
  DMLocalToGlobalBegin(da,l,ADD_VALUES,g);
  DMLocalToGlobalEnd(da,l,ADD_VALUES,g);
}
```

Listing 3: sf_dmda benchmark loop

3D grid of size `[X,Y,Z]` with nodes sequentially numbered in the *x, y, z* order. Suppose that within it is a subgrid of size `[dx,dy,dz]` with an index of the first node being `start`. The indices of the subgrid can be enumerated with `start+X*Y*k+X*j+i`, for `(i,j,k)` in `(0≤i<dx,0≤j<dy,0≤k<dz)`. Using this, the interior region of a local vector on this `DM` can be described as a subgrid of size `[n,n,1]` in a grid of size `[n+2,n+2,1]` with a start index `n+3`. Each face of the halo region can be described similarly. With this optimization, we need only to copy these grid parameters to the GPU; we then can easily calculate the needed indices there.

Since indices of ghost points are not contiguous, `PetscSF` has to allocate separate send and receive buffers and call the `pack` and `unpack` kernels, producing a code path similar to that in Figure 6(b) except that in the current case a `scatter` kernel is launched after `MPI_Irecv()` to do local communication. We perform back-and-forth communication between a global vector and a local vector using the code in Listing 3.

Note that in `DMLocalToGlobal()` we use `ADD_VALUES` instead of `INSERT_VALUES` since points along subgrid boundaries are reduced with ghost point values received from their neighbors. Consequently `PetscSF` has to handle potential data race conditions in the `unpack` kernel. We tested the code on Summit with two configurations. The first had nine compute nodes and one MPI rank per node. Since there was only inter-node communication, ideally all ranks should run uniformly with the same amount of communication and time. The other configuration had three compute nodes with three MPI ranks per node. MPI ranks were distributed in a packed manner such that ranks 0, 1, and 2 were on node 0; ranks 3, 4, and 5 were on node 1; and so forth. We placed each group of three ranks on one socket of a node. From Figure 12, we know that every rank did intra-socket communication with its eastern/western neighbors and did inter-node communication with its southern/northern neighbors. However, all ranks had even work and communication. Similar to the Ping-Pong test, we measured the average one-way latency of the communication, which is shown in Table 4.

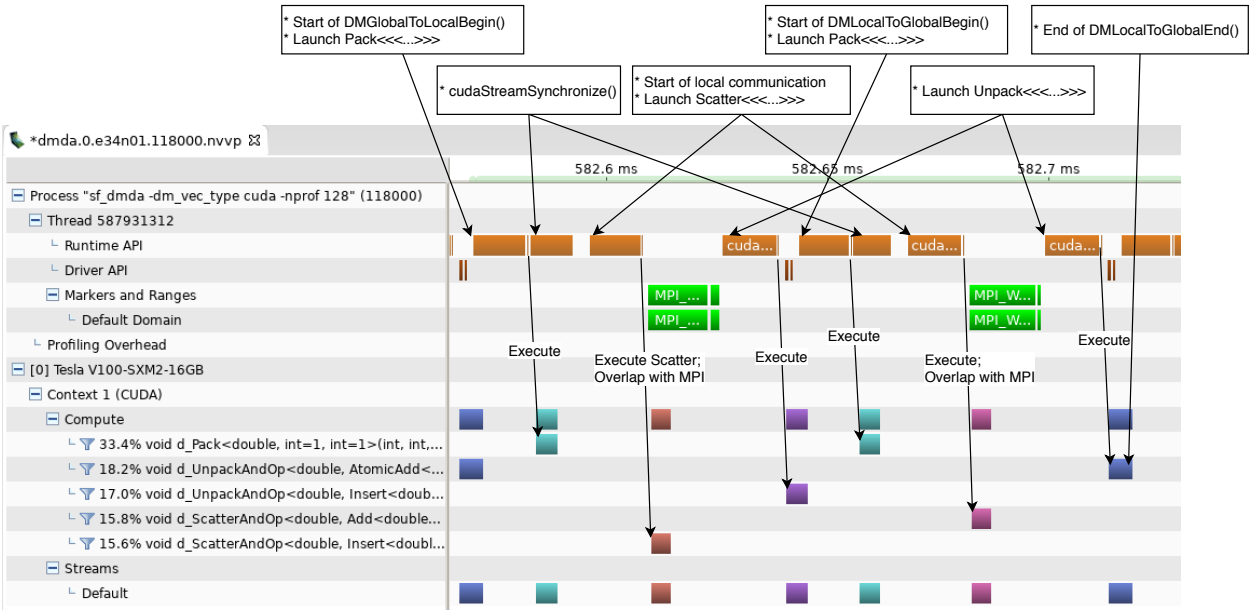

Figure 13: Timeline of one iteration of sf_dmda on rank 0 with nine nodes and $n = 128$

Table 4: One-way latency for the sf_dmda test, where n is the subgrid size and message size = 8n, which is the size of messages between two neighbors.

n	Message size (bytes)	Latency(µs)	
		Nine nodes	Three nodes
4	32	45.6	75.7
8	64	44.8	75.6
16	128	45.5	75.7
32	256	45.5	75.8
64	512	45.0	75.8
128	1K	46.0	75.9
256	2K	46.3	75.9
512	4K	47.1	76.0
1024	8K	57.1	83.0
2048	16K	139.9	139.0
4096	32K	499.9	498.3

We can see from the table that for small messages ($n \leq 512$) the latency is almost the same, which indicates that the MPI latency and CUDA run-time overhead dominates. Since the intra-socket Ping-Pong latency is longer than the inter-node one, the three-node configuration has a longer latency than the nine-node configuration has. Figure 13 shows profiling results of rank 0 with the nine-node configuration. We can see that MPI communication time is longer than the `scatter` kernel execution time and that the `pack` and `unpack` kernel launch times are prominent. In contrast, with larger n, the `scatter` kernel execution time, which is proportional to n^2, outweighs all times so that three nodes have the same execution time as nine nodes have. We can easily see this from the profiling result with $n = 4096$ in Figure 14.

5.4 PetscSF in irregular neighborhood communication

We now turn our attention to irregular communications. To study this problem, we use PETSc's sparse matrix-vector multiplication routine `MatMult(M,x,y)`, which calculates $y = Mx$. In PETSc, the sparse matrix is distributed by row, and the vectors x and y are also distributed accordingly. On each process, the local matrix is split into a diagonal submatrix A and an off-diagonal submatrix B. The multiplication with the diagonal part, Ax, needs to access only local entries of x and does not need communication, while

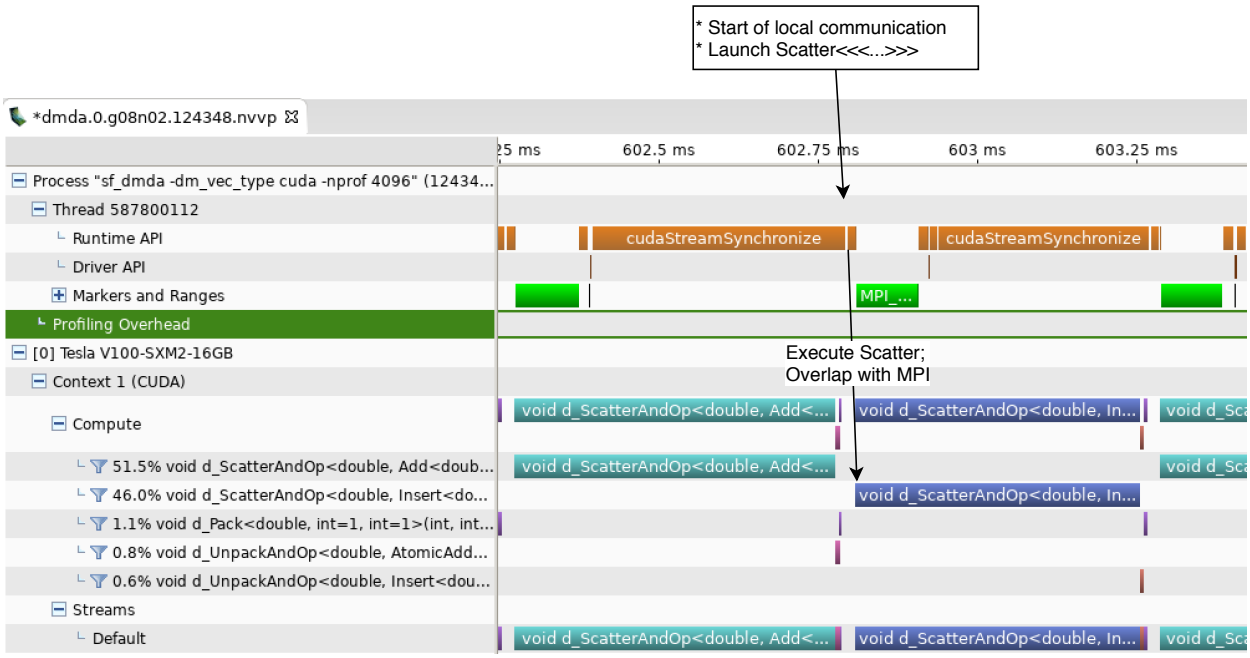

Figure 14: Timeline of one iteration of sf_dmda on rank 0 with n=4096

```
for (i=0; i<niter; i++) {
  VecScatterBegin(Mvctx,x,lvec,INSERT_VALUES,SCATTER_FORWARD);
  MatMult(A,x,y); /* overlapped computation: y = Ax */
  VecScatterEnd(Mvctx,x,lvec,INSERT_VALUES,SCATTER_FORWARD);
  MatMultAdd(B,lvec,y,y); /* y += B lvec */
}
```

Listing 4: MatMult benchmark loop

the multiplication with the off-diagonal part, Bx, needs to access remote entries of `x` and hence requires communication. The communication is done by `VecScatter`, which is implemented in `PetscSFBcast()`. In the matrix-vector product implementation, PETSc allocates a local vector lx working as star-forest leaves on each process to store remote entries of x. Without going into too many details, we have: the following. (1) The leaves are contiguous such that the `PetscSF` can directly use the leafdata (i.e., data array of $lvec$) as the leaf buffer in `PetscSFBcast()`, without resorting to an `unpack` kernel. (2) Since the matrix is sparse, each rank only needs to send out some entries of vector x (i.e., the roots). Therefore the roots are not contiguous, and we need a `pack` kernel. (3) There is no local communication. (4) The local computation, namely, `Ax`, could be overlapped with the communication. With that, we have the classical `MatMult(M,x,y)` implementation in PETSc, shown as the loop body in Listing 4 and in Figure 15(a).

From Figure 15(a), we see that the `cudaStreamSynchronize()` in `VecScatterBegin()` is only to ensure that `sbuf`, the output of the `unpack` kernel, is ready for use in `MPI_Isend()`. However, it accidentally blocks the launch of $y = Ax$, which is done through a cuSPARSE kernel. In other words, the launch cost of $y = Ax$ cannot be hidden. A remedy is to use CUDA events and rearrange `VecScatterBegin/End()`, as shown in Figure 15(b). There we record a CUDA event right after the pack and move `MPI_Isend()` from `VecScatterBegin()` to `VecScatterEnd()`. The event is synchronized before `MPI_Isend()` so that MPI does not send out incorrect data. Note that the $Blvec$ in Figure 15(b) depends only on the communication results and does not depend on $y = Ax$. However, the algorithm requires $y = y + Blvec$ to be executed after $y = Ax$. We can decouple this dependency with the help of a temporary vector `z`. In Figure 15(c), we launch $z = Blvec$ on a new stream s and then launch kernel $y = y + z$ on the default stream to add the partial result to `y`. We use CUDA events to build the dependency between the two kernels on different streams. As long as the communication finishes before kernel $y = Ax$, kernel $z = Blvec$ has the potential to run concurrently

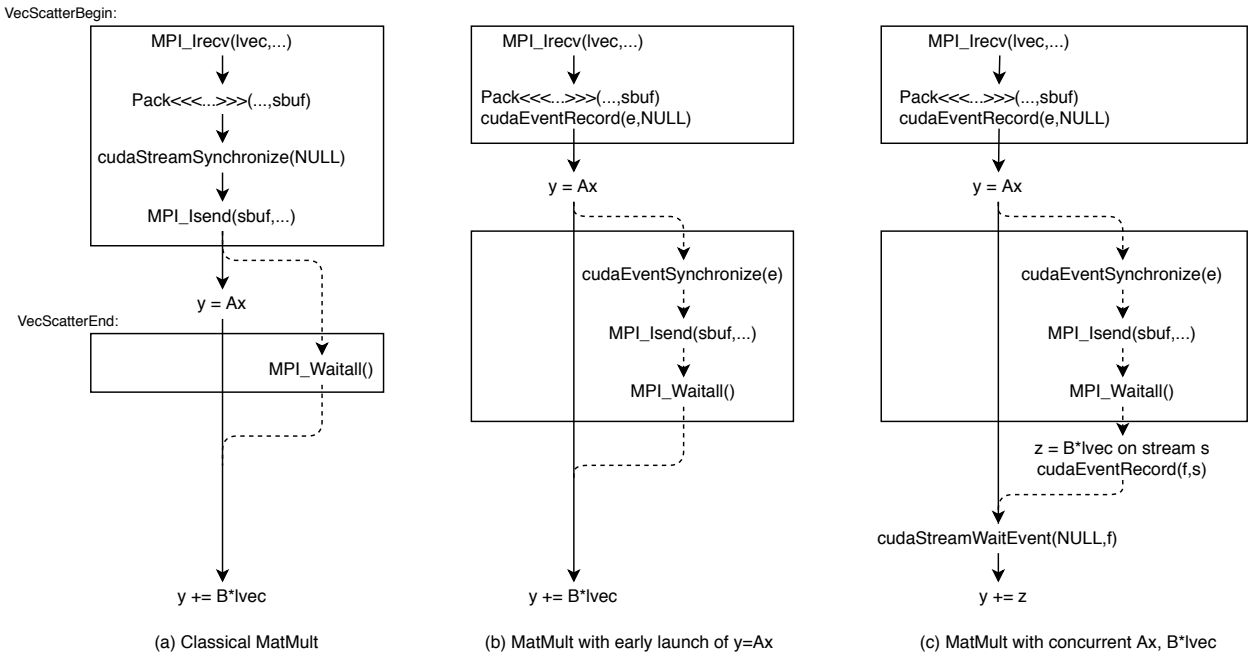

Figure 15: Various matrix-vector product implementations. Boxes at the top are the `VecScatterBegin()`, and those at the bottom are the `VecScatterEnd()`. In each diagram, vertically parallel solid and dashed lines indicate overlapped computation and communication.

with $y = Ax$. Since $y = Ax$ and $y = y + z$ are both launched on the default stream, their dependency is automatically maintained. Note that both Figures 15(b) and (c) assume that the computation sandwiched between `VecScatterBegin/End()` will not block the CPU thread so that `MPI_Isend()` can be posted as soon as possible. Therefore, without changes, they cannot be directly applied to CPU codes. They are currently not in the PETSc release.

We tested these three matrix-vector product implementations with a sparse matrix (HV15R) from the Florida sparse matrix collection [4]. The size of the matrix is 2,017,169, and it has 283,073,458 nonzeros. For the three matrix-vector productstTested on one node of Summit with six GPUs and six MPI ranks, the execution times were 918.9µs, 902.2µs, and 904.6µs, implementations, respectively. We can see that MatMult(b) was 16.7µs faster than MatMult(a), which is close to a kernel launch time, indicating that the launch time of $y = Ax$ is effectively hidden in MatMult(b). However, MatMult(c) did not show an advantage over MatMult(b). We show their timelines on rank 3 in Figures 16 and 17. We can see that the sparse matrix-vector products (i.e., csrMv_kernel) with the diagonal block and the off-diagonal block did overlap as we expected. However, we also found that with overlapping, the kernel's execution time was a little longer than without overlapping, offsetting any gains from overlapping. Further investigation revealed the reason. In CUDA, concurrent kernel execution requires that there be enough resources to accommodate multiple kernels. Neither kernel can have enough resident thread blocks to fill up the GPU. Moreover, a streaming multiprocessor (SM) can host thread blocks only from the same kernel. In our test, kernel $y = Ax$ had a grid of size (42025,1,1) and a thread block of size (16,8,1), while kernel $z = Blvec$ had a grid of size (10507,1,1) and a thread block of size (4,32,1) (note that the cuSPARSE library controls these kernel launch parameters). However, the NVIDIA V100 GPU has 80 SMs, and each SM can have only 32 resident thread blocks, giving a total of 2,560 resident thread blocks per GPU. Therefore, we saw an overlap only at the end of the first kernel; presumably this was when some SMs were draining from the first kernel and became available for the second one. Additionally, since sparse matrix-vector products are memory bandwidth bound, running two sparse matrix-vector products concurrently limits only the memory bandwidth available to each and hurts their performance. We predict that small compute-bound kernels would benefit from the design in Figure 15(c).

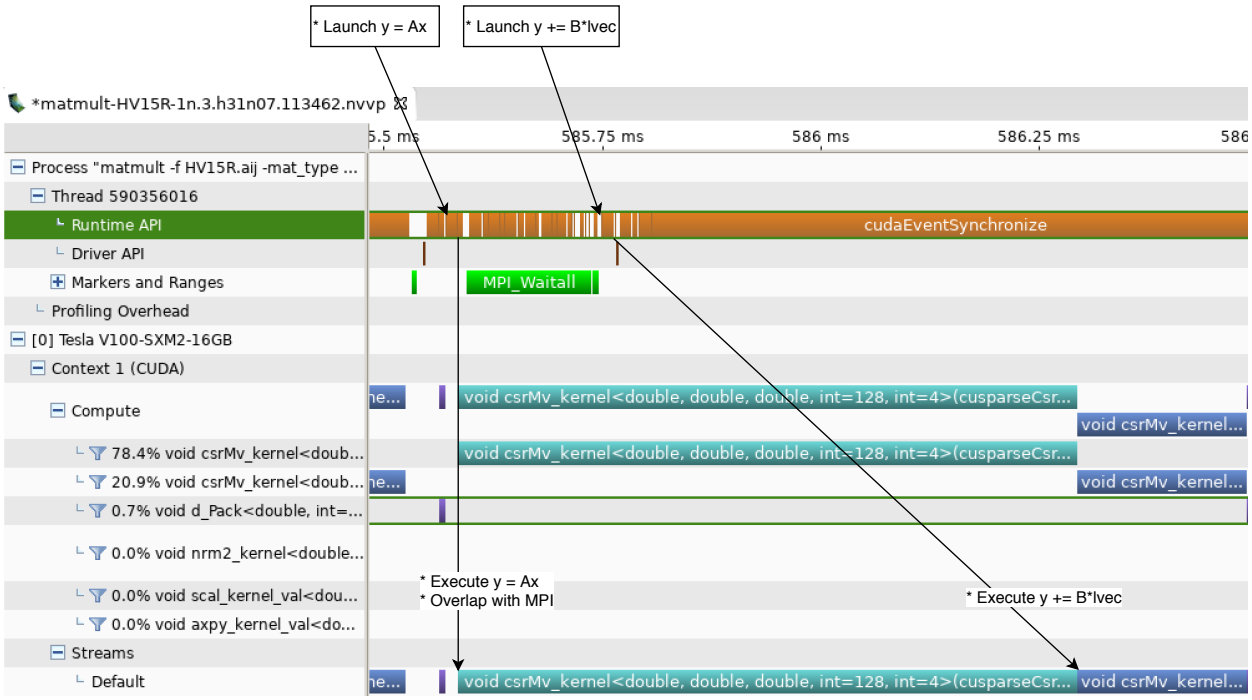

Figure 16: Timeline of matrix-vector product with early launch of $y = Ax$. Note that the launch of kernel $y = Ax$ does not need to wait for the `pack` kernel to complete, but the kernel $y = y + Blvec$ cannot start until the kernel $y = Ax$ has completed.

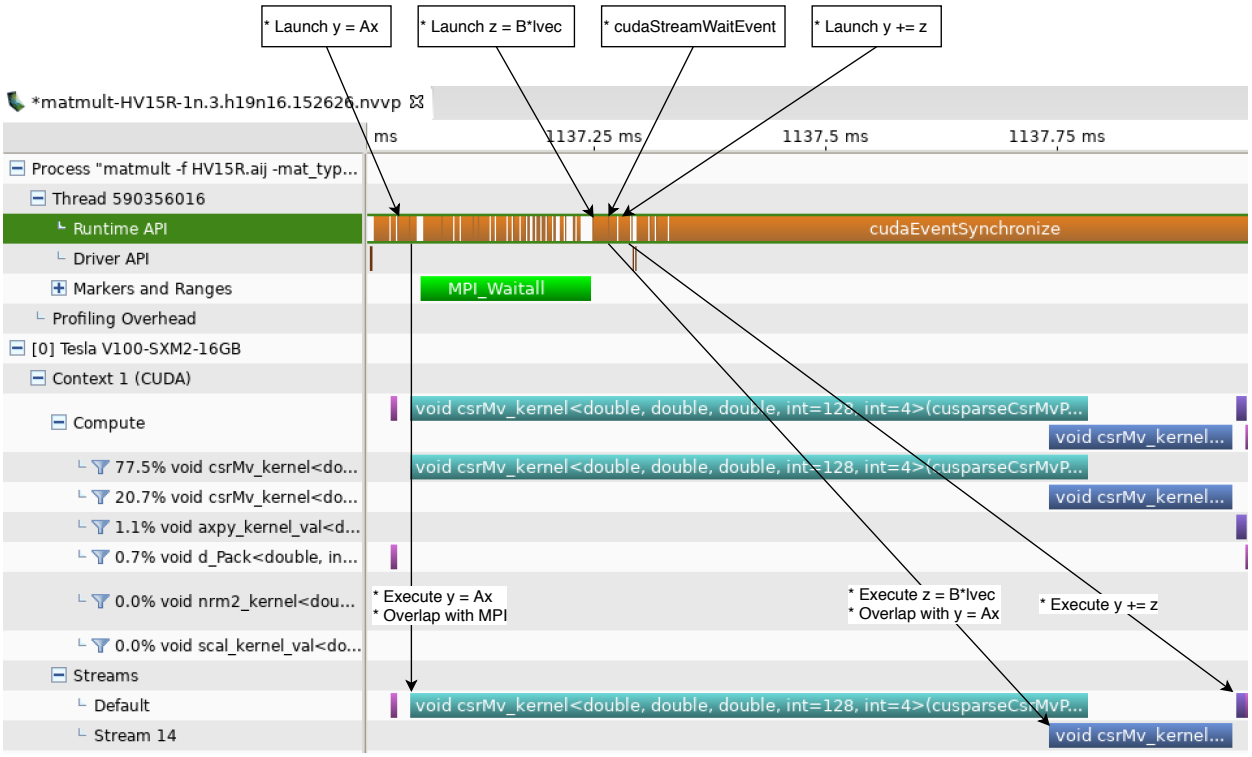

Figure 17: Timeline of matrix-vector product with concurrent kernels Ax and $Blvec$

6 Discussion and Conclusion

Asynchronous computation on GPUs brings new challenges to MPI communication. The application or software library's communication module has to synchronize the device correctly and also provide efficient `pack/unpack` kernels. In this report, we analyzed and tested `PetscSF`, the communication module in PETSc, on Summit GPUs. We first measured GPU communication latencies with an MPI Ping-Pong benchmark, which does not have any synchronizations or `ppack/pack` kernels, and therefore this performance provides an upper bound for that of `PetscSF`. In Section 4 we introduced three synchronization models in `PetscSF`. In Section 5.1 we evaluated a Ping-Pong test (sf_pingpong) written in `PetscSF` under those models. From the test results, we know the costs of the various CUDA synchronizations. We found that the extra overhead introduced by `PetscSF` can be as low as 1µs. In Section 5.2 we introduced two new benchmarks (sf_unpack and sf_scatter) that have unpacking and local communication. In these benchmarks we measured the kernel launch cost and the effect of overlapping local and remote communication. In Section 5.3 we introduced index optimizations in `pack` and `unpack` kernels with regular neighborhood communication. In this communication pattern, with small (regular) domains, remote communication is the bottleneck, and with large domains, local communication is the bottleneck. In Section 5.4 we evaluated `PetscSF`'s performance on irregular neighborhood communication with a sparse matrix-vector multiplication kernel.

`PetscSF`'s default synchronization model assumes that the input data and output data are on the default stream, so that we can avoid the `cudaDeviceSynchronize()` and `cudaStreamSynchronize()` calls before the `pack` kernel and after the `unpack` kernel, respectively, an action that translates into a savings of 9µs. The remaining synchronization is a `cudaStreamSynchronize()` call, which costs about 4µs and is denoted by $T_{StreamSync}$. With that, we can model the total time T of a general split-phase communication pattern `PetscSFXxxBegin(); userkernel(); PetscSFXxxEnd()` as follows.

$$T = T(pack) + T_{StreamSync} + \max \left\{ \begin{matrix} l_{MPI}, \\ T(scatter) \oplus T(userkernel) \end{matrix} \right\} \oplus T(unpack) \tag{4}$$

Here $T(K)$ represents the time of kernel K, including the launch and execution time, and l_{MPI} is the MPI communication time. Again $\oplus$ indicates that the next kernel's launch time could be overlapped with the execution of the previous kernel.

The `pack`, `unpack`, and `scatter` kernels involve only simple operations on the elements (i.e., roots or leaves) and are usually bandwidth bound. One can easily model their execution time as $startuptime + \frac{memory\ size}{effectivebandwidth}$, where memory size is the total size of data that the kernel accesses, including both the values and their indices if the memory access is irregular. The effective bandwidth depends on the access pattern, which may be contiguous, strided, or random. One can write simple benchmarks to measure each of these scenarios. The startup time is the time required to launch a CUDA kernel unless that time can be hidden by currently running kernels. For point-to-point communication involving only a pair of ranks, one can easily model l_{MPI} with Eq. 1 in Section 3, and one could validate Eq. 4 using data from Sections 5.1 and 5.2. For communication involving multiple senders and receivers sharing communication links, we do not have a reliable model. LogGP[1] might be an alternative, but we do not know how to validate it on Summit. We leave this as an open question.

$T_{StreamSync}$ of 4µs seems not too high; however, the synchronization may block further kernel launches in the pipeline, resulting in poor kernel launch time hiding, which could result in a time much higher than the time of `cudaStreamSynchronize()` itself. For example, assume that we have five kernels A, B, C, D, and E and that their execution time is 40µs, 5µs, 5µs, 5µs, and 5µs, respectively. Let us further assume that a kernel launch costs 10µs. If kernel launches are fully pipelined, the total time for these five kernels is 70µs, as shown in Figure 18(a). However, if there is a `cudaStreamSynchronize()` after the first kernel launch, then the remaining launches will stall, and the total time will be 95µs, as shown in FIgure 18(b).

In Section 5.4 we introduced an approach that uses CUDA events to avoid having to call `cudaStreamSynchronize()`, but this approach requires asynchronous operations between `VecScatterBegin()` and `VecScatterEnd()`, and there cannot be too many such operations since we need to issue `MPI_Isend()` as soon as possible. The ideal solution would be to make MPI routines CUDA stream aware, such that a nonblocking MPI call worked as an asynchronous kernel launch on a given stream and an `MPI_Wait()` worked as a `cudaEventSynchronize()`. In this way, MPI calls would be regular nodes in the computation dependence graph, instead of a barrier in it.

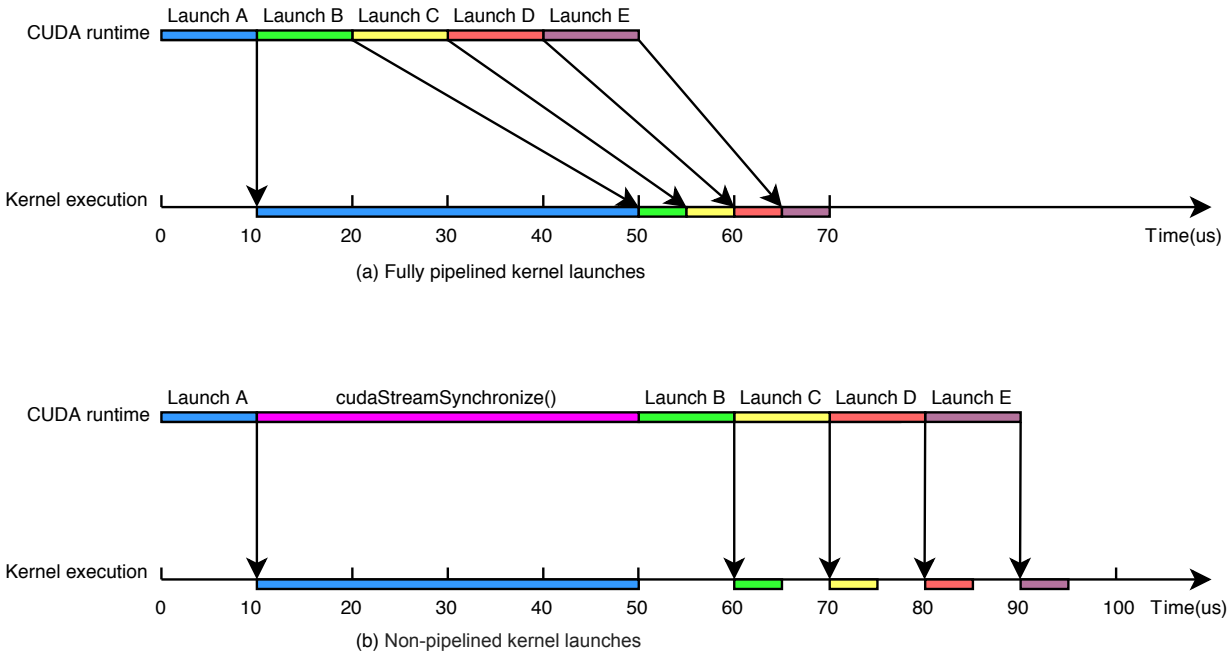

Figure 18: Effect of synchronization in kernel launches. Without synchronization, the five kernels from A to E take 70µs to finish. With a single `cudaStreamSynchronize()`, they take 95µs.

Acknowledgments

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the logP model for parallel computation. Journal of parallel and distributed computing, 44(1):71–79, 1997.

[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Rev 3.13, Argonne National Laboratory, 2020.

[3] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page, 2020. http://www.mcs.anl.gov/petsc.

[4] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS), 38(1):1–25, 2011.

[5] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen. Aluminum: An asynchronous, GPU-aware communication library optimized for large-scale training of deep neural networks on HPC systems. In 2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC), pages 1–13, Nov 2018.

[6] Judy Hill. Summit at the Oak Ridge Leadership Computing Facility, 2018. `https://press3.mcs.anl.gov//atpesc/files/2018/08/ATPESC_2018_Track-1_6_7-30_130pm_Hill-Summit_at_ORNL.pdf`.

[7] Kawthar Shafie Khorassani, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Performance evaluation of MPI libraries on GPU-enabled OpenPOWER architectures: Early experiences. In International Conference on High Performance Computing, pages 361–378. Springer, 2019.

[8] Hannah Morgan, Richard Trans Mills, and Barry Smith. Evaluation of PETSc on a heterogeneous architecture, the OLCF Summit system: Part I – vector node performance. Technical Report ANL-19/41, Mathematics and Computer Science Division, Argonne National Laboratory, 2019.

[9] DK Panda et al. OSU Microbenchmarks v5.6.2. http://mvapich.cse.ohio-state.edu/benchmarks/, 2019.

[10] Exascale Support Team. Exascale web page, 2019. https://exascaleproject.org/.

[11] Summit Support Team. Summit User Guide. https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/. Accessed: 2019-08.

[12] Nicholas Wilt. The CUDA handbook: A comprehensive guide to GPU programming. Pearson Education, 2013.

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC